Measuring Code Coverage with JaCoCo

▼ Setting Up Code Coverage with JaCoCo

1. **Project Setup**:
   - Clone the `student-exam-registry` app from GitHub
   - Ensure your `pom.xml` includes the **JaCoCo plugin**.

   Example:

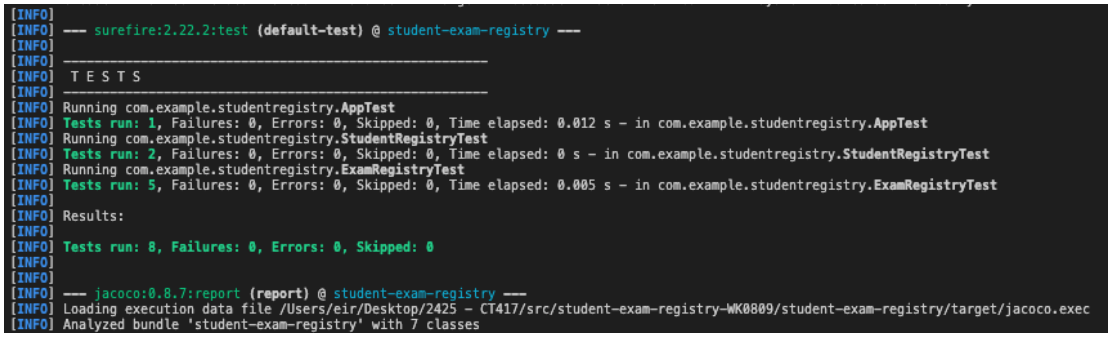
```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.7</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
```

```
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

2. **Run the Code Coverage Analysis**:
    - Execute the following Maven command:

```
mvn clean test
```

    - JaCoCo will automatically generate a coverage report after the tests have run.
3. You don't have to do anything special after adding the plugin—code coverage will run with every test execution.

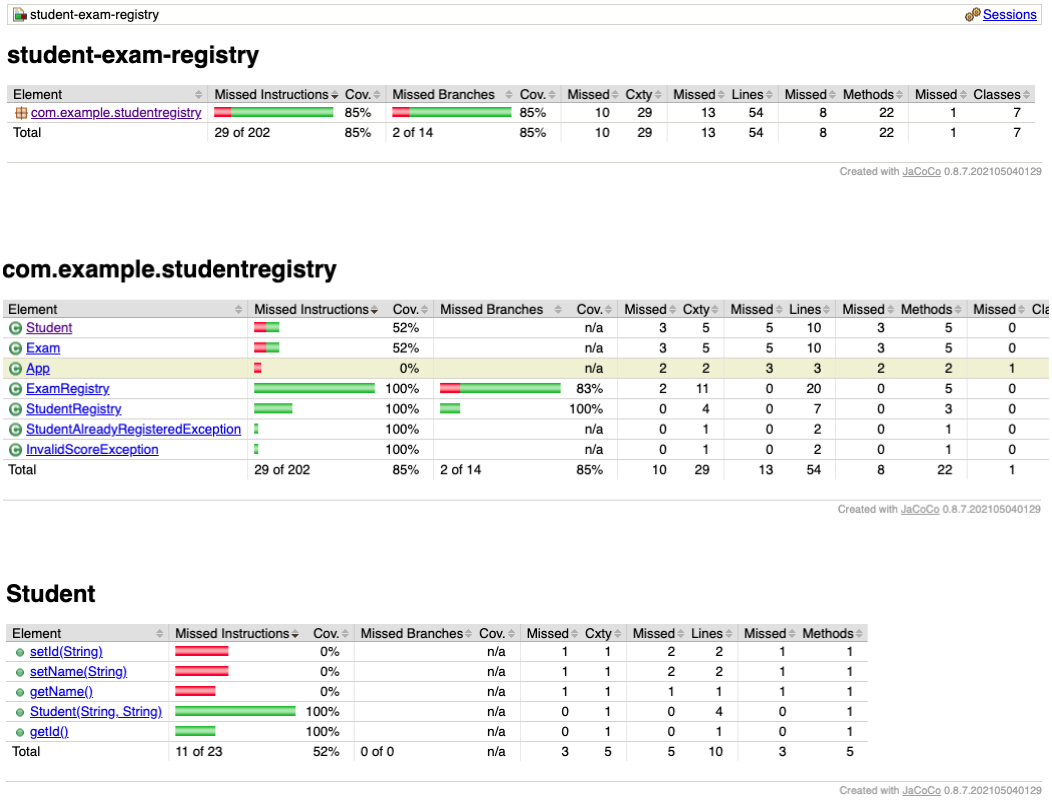


▼ Accessing and Interpreting the JaCoCo Report

1. **Viewing the Report**:
    - After the tests complete, navigate to:

```
target/site/jacoco/index.html
```

    - Open the `index.html` file in a browser to view the coverage results.



student-exam-registry

Sessions

student-exam-registry

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.studentregistry		85%		85%	10	29	13	54	8	22	1	7
Total	29 of 202	85%	2 of 14	85%	10	29	13	54	8	22	1	7

Created with JaCoCo 0.8.7.202105040129

com.example.studentregistry

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Cla
Student		52%		n/a	3	5	5	10	3	5	0	
Exam		52%		n/a	3	5	5	10	3	5	0	
App		0%		n/a	2	2	3	3	2	2	1	
ExamRegistry		100%		83%	2	11	0	20	0	5	0	
StudentRegistry		100%		100%	0	4	0	7	0	3	0	
StudentAlreadyRegisteredException		100%		n/a	0	1	0	2	0	1	0	
InvalidScoreException		100%		n/a	0	1	0	2	0	1	0	
Total	29 of 202	85%	2 of 14	85%	10	29	13	54	8	22	1	

Created with JaCoCo 0.8.7.202105040129

Student

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
setId(String)		0%		n/a	1	1	2	2	1	1
setName(String)		0%		n/a	1	1	2	2	1	1
getName()		0%		n/a	1	1	1	1	1	1
Student(String, String)		100%		n/a	0	1	0	4	0	1
getId()		100%		n/a	0	1	0	1	0	1
Total	11 of 23	52%	0 of 0	n/a	3	5	5	10	3	5

Created with JaCoCo 0.8.7.202105040129

2. **Interpreting the Coverage Report**:
   - **Instruction Coverage** = Percentage of `bytecode` instructions executed.
   - **Branch Coverage** = Percentage of control structures (e.g., `if-else`, `switch`) covered.
   - **Complexity** = Measures the cyclomatic complexity of the code (lower complexity is easier to maintain).
   - **Missed Lines** = Lines of code not covered by tests.

▼ Analysing the Coverage Results

1. **Understanding the Results**:
   - **For e.g., 85% Code Coverage** (Is it good?)
     - 85% of the instructions are covered by the current test suite.
     - Uncovered branches indicate areas where the logic is not fully tested.
     - **High coverage** is a good sign, but it doesn't always mean the code is perfect.

2. **Improvement Opportunities**:
    - **Missed Branches or Lines**: Use this data to identify untested parts of your code.
    - Consider **refactoring** overly complex code (high cyclomatic complexity) to improve maintainability and testability.

▼ Add Unit Tests for `Null` and Invalid Conditions

We wrote the following tests to cover those missing conditions:

1. **Test for `Null` Inputs in `StudentRegistry` and `ExamRegistry`**
    - These tests ensure that `IllegalArgumentException` is thrown if `null` values are passed as inputs for students or exams.

```
// StudentRegistry
@Test
void register_NullStudent_ShouldThrowException() {
    Exception exception = assertThrows(IllegalArgumentExc
        studentRegistry.register(null);
    });
    assertEquals("Student must not be null.", exception.g
}
```

```
// ExamRegistry
@Test
void enrol_NullStudent_ShouldThrowException() {
    Exception exception = assertThrows(IllegalArgumentExc
        examRegistry.enrol(null, exam);
    });
    assertEquals("Student and Exam must not be null.", ex
}

@Test
```

```
void enrol_NullExam_ShouldThrowException() {
    Exception exception = assertThrows(IllegalArgumentExc
        examRegistry.enrol(student, null);
    });
    assertEquals("Student and Exam must not be null.", ex
}
```

2. **Test for Handling Invalid Exam Scores**
   - We ensured that an `InvalidScoreException` is thrown if the score is not between 0 and 100.

```
@Test
void recordScore_InvalidScore_ShouldThrowException() {
    examRegistry.enrol(student, exam);
    Exception exception = assertThrows(InvalidScoreExc
eption.class, () -> {
        examRegistry.recordScore(student, exam, 150);
// Invalid score
    });
    assertEquals("Score must be between 0 and 100.", e
xception.getMessage());
}
```

3. We ran `mvn clean test` to execute the updated unit tests.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
Student		52%		n/a
Exam		52%		n/a
App		0%		n/a
ExamRegistry		100%		87%
StudentRegistry		100%		100%
StudentAlreadyRegisteredException		100%		n/a
InvalidScoreException		100%		n/a
Total	29 of 218	86%	2 of 20	90%

- The updated report now shows 86**% code coverage**.
- The missing branches were covered, and we focused on handling edge cases that JaCoCo highlighted as gaps.

▼ Pushing Code Coverage to 100% — Critical Fixes

- Critical fixes focus on adding tests for logic-heavy sections of the code, such as conditional branches, exception handling, and validation logic.
- These are the areas where code coverage is most important as they directly impact the reliability and behaviour of the application.

1. **Identify Missing Branches**:

   The JaCoCo report shows that certain branches of the code in `ExamRegistry` and `StudentRegistry` are not fully tested, especially in cases involving exceptions or edge conditions.

   Example Missing Branches:

   - Null checks in `ExamRegistry.enrol()`.
   - Edge case handling in `StudentRegistry.register()`.

2. **Write Tests for Missing Branches**:

   Add tests that cover these branches by forcing the code to hit every conditional path.

   Example: `ExamRegistryTest.java`

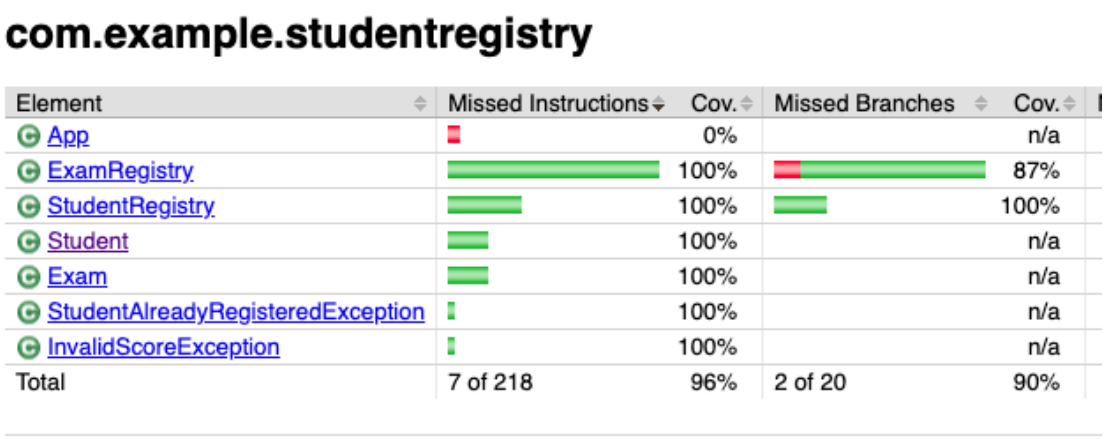
```
@Test
void enrol_NullExam_ShouldThrowException() {
    Exception exception = assertThrows(IllegalArgumentExc
        examRegistry.enrol(student, null);
    });
    assertEquals("Student and Exam must not be null.", ex
}

@Test
void enrol_NullStudent_ShouldThrowException() {
    Exception exception = assertThrows(IllegalArgumentExc
        examRegistry.enrol(null, exam);
    });
    assertEquals("Student and Exam must not be null.", ex
}
```

   Example: `StudentRegistryTest.java`

```
@Test
void register_NullStudent_ShouldThrowException() {
```

```
    Exception exception = assertThrows(IllegalArgumentExc
        studentRegistry.register(null);
    });
    assertEquals("Student must not be null.", exception.g
}
```

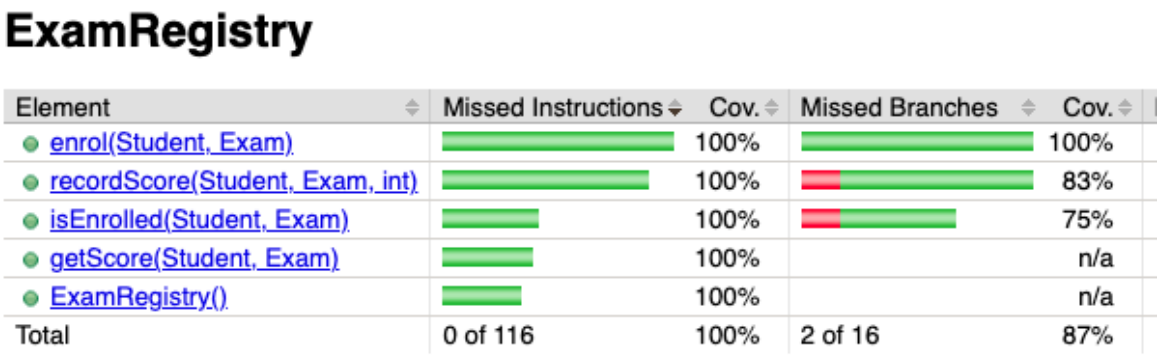
3. **Run Tests and Verify the Coverage**:



com.example.studentregistry

Element	Missed Instructions	Cov.	Missed Branches	Cov.
App		0%		n/a
ExamRegistry		100%		87%
StudentRegistry		100%		100%
Student		100%		n/a
Exam		100%		n/a
StudentAlreadyRegisteredException		100%		n/a
InvalidScoreException		100%		n/a
Total	7 of 218	96%	2 of 20	90%

▼ Pushing Code Coverage to 100% — Critical Fixes



ExamRegistry

Element	Missed Instructions	Cov.	Missed Branches	Cov.
enrol(Student, Exam)		100%		100%
recordScore(Student, Exam, int)		100%		83%
isEnrolled(Student, Exam)		100%		75%
getScore(Student, Exam)		100%		n/a
ExamRegistry()		100%		n/a
Total	0 of 116	100%	2 of 16	87%

- **ExamRegistry** still has **missed branches** in its logic, even though the instructions are fully covered.
- Specifically, the branches within the `isEnrolled` and `recordScore` methods have missed some scenarios that need additional test coverage.

  Here's how we can address these issues:

  1. `isEnrolled` **Method:**

  The **75% branch coverage** indicates there's a conditional flow that hasn't been fully tested. Let's add a few more to cover all permutations.

```
// Additional test for isEnrolled method
    @Test
    void isEnrolled_StudentNotEnrolled_ShouldReturnFalse(
        Exam newExam = new Exam("ENG102", "English 102");
        assertFalse(examRegistry.isEnrolled(student, newE
    }

    @Test
    void isEnrolled_ExamNotExist_ShouldReturnFalse() {
        Exam nonExistentExam = new Exam("SCI203", "Scienc
        assertFalse(examRegistry.isEnrolled(student, nonE
    }

    @Test
    void isEnrolled_EmptyEnrollmentList_ShouldReturnFalse
        Exam newExam = new Exam("HIST203", "History 203");
        assertFalse(examRegistry.isEnrolled(student, newE
    }

    @Test
    void isEnrolled_EmptyExamList_ShouldReturnFalse() {
        // Ensure there's no entry for this exam
        Exam newExam = new Exam("MATH104", "Mathematics 1
        assertFalse(examRegistry.isEnrolled(student, newE
    }

    @Test
    void isEnrolled_StudentEnrolled_ShouldReturnTrue() {
        examRegistry.enrol(student, exam); // Enroll the
        assertTrue(examRegistry.isEnrolled(student, exam)
    }

    @Test
    void isEnrolled_ExamWithoutStudents_ShouldReturnFalse
        // No enrollment list for this exam
        assertFalse(examRegistry.isEnrolled(student, exam
    }
```

```
    @Test
    void isEnrolled_OtherStudentsEnrolled_ShouldReturnFal
        Exam newExam = new Exam("ENG102", "English 102");
        // Enroll another student in this new exam
        examRegistry.enrol(new Student("S002", "Bob"), ne
        // Check if the original student is enrolled (the
        assertFalse(examRegistry.isEnrolled(student, newE
    }

    @Test
    void isEnrolled_StudentIsEnrolled_ShouldReturnTrue()
        examRegistry.enrol(student, exam);
        assertTrue(examRegistry.isEnrolled(student, exam)
    }

    @Test
    void isEnrolled_EmptyExam_ShouldReturnFalse() {
        // Check that the student is not enrolled in a ne
        Exam newExam = new Exam("HIST203", "History 203");
        assertFalse(examRegistry.isEnrolled(student, newE
    }
```

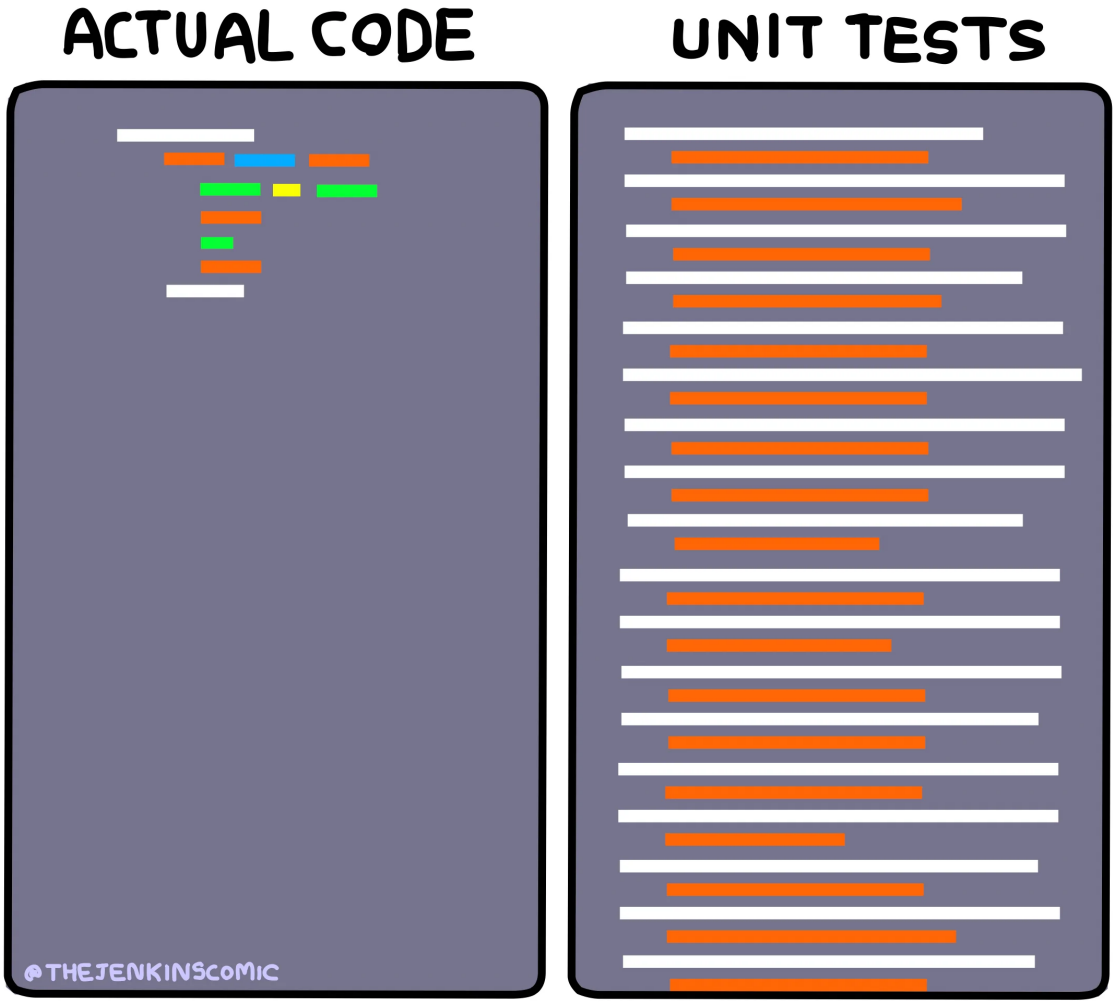
2. `recordScore` **Method:**

The **83% branch coverage** suggests there's still a path where the method is called without the necessary conditions (e.g., without the student being enrolled). Let's add a few more tests to cover all the permutations.

```
// Additional test for recordScore method
    @Test
    void recordScore_ZeroAndMaxScore_ShouldWork() throws
        examRegistry.enrol(student, exam);

        // Test for lower boundary (0)
        examRegistry.recordScore(student, exam, 0);
        assertEquals(0, examRegistry.getScore(student, ex

        // Test for upper boundary (100)
        examRegistry.recordScore(student, exam, 100);
        assertEquals(100, examRegistry.getScore(student,
```

```
    }

    @Test
    void recordScore_ValidBoundaryScores_ShouldWork() thr
        // Enroll the student
        examRegistry.enrol(student, exam);

        // Test boundary value of 0
        examRegistry.recordScore(student, exam, 0);
        assertEquals(0, examRegistry.getScore(student, ex

        // Test boundary value of 100
        examRegistry.recordScore(student, exam, 100);
        assertEquals(100, examRegistry.getScore(student,
    }

    @Test
    void recordScore_NegativeScore_ShouldThrowException()
        // Enroll the student
        examRegistry.enrol(student, exam);

        // Test negative score
        Exception exception = assertThrows(InvalidScoreEx
            examRegistry.recordScore(student, exam, -1);
        });
        assertEquals("Score must be between 0 and 100.",
    }
```

▼ Best Practices



- While achieving 100% coverage is not always necessary, this exercise demonstrates how focusing on the **critical code paths** (such as handling invalid inputs or edge cases) can significantly improve the code's robustness.
- The key classes handling business logic (`ExamRegistry` and `StudentRegistry`) were critical to achieving better coverage.
    - Focus on edge cases that directly impact functionality, like boundary value testing and ensuring all control flow branches are covered.
- While testing trivial getters/setters (as in `Student` and `Exam` classes) is possible, our focus was on covering logic that directly affects application behaviour.
    - Although, non-functional classes (e.g., simple data holders) can be covered for completeness, they are not essential to the core functionality.
